Data Mining II, C.A. Brebbia & N.F.F. Ebecken (Editors)
© 2000 WIT Press, www.witpress.com, ISBN 1-85312-821-X

Modeling dynamical systems by recurrent neural networks

H.G. Zimmermann & R. Neuneier
Information and Communications, Corporate Technology, Siemens AG, Germany.

Abstract

We present our experiences of time series modeling by finite unfolding in time. The advantage of this approach is that the set of learnable neural network functions is restricted by a set of regularization methods which do not constrain the essential dynamics. Keywords in this section are over- and undershooting, the analysis of cause and effect, and the estimation of the embedding dimension in a partially externally driven dynamic system.

1 Introduction

Standard analysis of dynamic systems by feedforward neural networks translates the time series identification problem into a pattern recognition approach. Typically, the first step in such an analysis looks for an appropriate description of the present time state which can be used as an input vector. This is usually very tricky because one has to consider many preprocessing techniques (necessary time lags, smoothing transformations, using derivatives of data, etc.). Then, the central identification problem is solved by relying on the universal approximation property that a sufficiently large three layer neural network can in principle model any continuous function on a compact domain [1]. However, if we use such a task independent approach, the characteristics of the available data determine the quality of the resulting model. We believe that this is a misleading point of view, especially if the amount of useful information that can be extracted from the data is small. For example, the amount of data is too small in comparison to its dimensionality, data is very noisy, available data does not sufficiently cover the input space and so forth. Instead of using simple pattern recognition, we propose a framework of recurrent networks which incorporate prior knowledge of the dynamic systems we want to model via extended network architectures.

Using the architecture in section 2, we propose a finite unfolding in time as an implementation for recurrent neural networks. Then, in section 3, we develop the *overshooting* technique which generates additional information useful for decision support systems and which also improves the learning. We discuss the consequences of overshooting on partially external driven dynamic systems. In section 4 we question if the time grid of the model has to be the same as the time grid of the data. The network of section 4 is a realization of a model which has a finer time grid than that of the observed data. This leads to important dynamic properties of the resulting model.

2 Representing dynamic systems by recurrent networks

The following set of equations (eqns (1)), consisting of a state and output equation, is a recurrent description of a dynamic system in a very general form for discrete time grids.

$$\begin{aligned} s_t &= f(s_{t-1}, u_t) && \text{state transition} \\ y_t &= g(s_t) && \text{output equation} \end{aligned} \tag{1}$$

The state transition is a mapping from the previous internal hidden state of the system s_{t-1} and the influence of external inputs u_t to the new state s_t. The output equation gives rise to the observable output vector y_t.

The identification task eqn (1) can be implemented as a *time-delay recurrent neural network*:

$$\begin{aligned} s_t &= NN(s_{t-1}, u_t; v) && \text{state transition} \\ y_t &= NN(s_t; w) && \text{output equation.} \end{aligned} \tag{2}$$

By specifying the functions f, g as neural networks with parameter vectors v, w we have transformed the task into a parameter identification problem:

$$\frac{1}{T}\sum_{t=1}^{T}\left(y_t - y_t^d\right)^2 \to \min_{v,w} \tag{3}$$

The dynamic system consisting of the two equations eqns (2) can be implemented as the one neural network architecture as shown Fig. 1. The weights are $v = \{A, B\}$

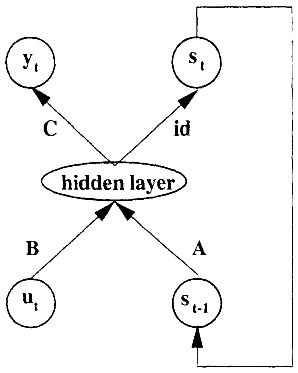

Figure 1: A time-delay recurrent neural network.

and $w = \{C\}$. In general one may think of a matrix D instead of the identity matrix (id) between the hidden layer and s_t. This is not necessary because for a linear output layer s_t we can combine matrices A and D to a new matrix A' between the input s_{t-1} and the hidden layer. Note that the output equation $NN(s_t; w)$ is realized as a linear function. It is straightforward to show by using an augmented inner state vector that this is not a functional restriction.

After these definitions the next section describes a specific neural network implementation of time delay recurrent networks.

Now we unfold the network of Fig. 1 over time using *shared weight matrices* A, B, C (Fig. 2). Shared weights share the same memory for storing their weights, i. e. the weight values are the same at each time step of the unfolding (see [2,3,4,5,6] for approaches which also share weights).

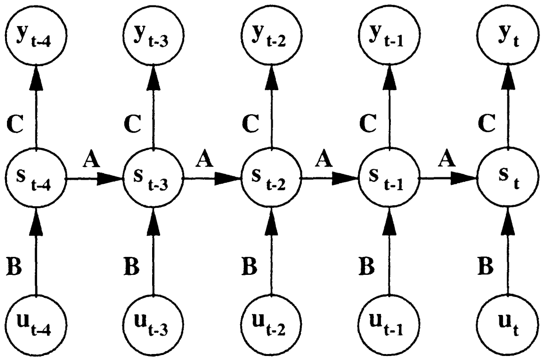

Figure 2: Finite unfolding realized by shared weights A, B, C.

The approximation step is the finite unfolding which truncates the unfolding after some time steps (for example, we choose $t-4$ in Fig. 2). The important question to solve is the determination of the correct amount of past information to include in the model of y_t. Typically, you start with a given truncation length (in our example we unfold up to $t-4$). Then you can observe the individual errors of the outputs $y_{t-4}, y_{t-3}, y_{t-2}, y_{t-1}$ and y_t computed by eqn (3) which usually are decreasing from left (y_{t-4}) to right (y_t). The reason for this observation is that the leftmost output y_{t-4} is computed only by the most past external information u_{t-4}, the next output y_{t-3} depends also on its external u_{t-3} but it uses the additional information of the previous internal state s_{t-4}. By such superposition of more and more information the error value will decrease until a minimal error is achieved. This saturation level indicates the maximum length of time steps which contribute relevant information to model the present time state.

Knowing e. g. that the saturation takes place at $t-1$ we achieve an unfolding with memory of past time steps $t-3, \cdots, t-1$ and the information of present time step t. Now, in order to achieve a consistent model we have to disconnect the internal flow of error signals generated by the outputs from $y_{t-3}, y_{t-2}, y_{t-1}$ (see Fig. 3). Otherwise, the shared weight matrices A, B, C would receive inappropriate gradient information due to different dependencies over time. For example, y_{t-3} only learns a mapping from u_{t-3} whereas y_t is trained on $(u_t, \cdots, u_{t-3})$.

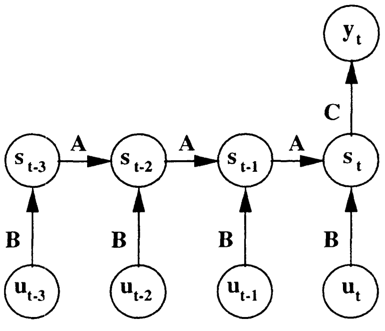

Figure 3: Finite unfolding assuming error level saturation within three time steps.

Having in mind that we are finally interested in forecasting we should ask ourselves what are the improvements of the modeling up to now. If y_t is a description of the shift of a variable (e. g. a price shift $y_t = p_{t+1}/p_t - 1$ in a financial forecast) we predict the target variable one step ahead. Thus, the result of the model in Fig. 3 is only a sophisticated preprocessing of the inputs $u_{t-3}, \cdots, u_t$ to generate a present time state description s_t from which the forecast y_t is computed. In contrast to typical feedforward neural networks whose success heavily depends on an appropriate, sometimes very complicated, preprocessing, our approach only needs a simple transformation of the raw inputs x_t in form of a scaled momentum as in eqn (4).

$$u_t = \text{scale}\left(\frac{x_t - x_{t-1}}{x_{t-1}}\right) \tag{4}$$

This releases us from smoothing the data or computing first and higher order moments of the shift variables.

A further advantage of the recurrent preprocessing (Fig. 3) is the moderate usage of free parameters. In a feedforward neural network an expansion of the delay structure leads automatically to an increase of the number of weights. In the recurrent formulation (Fig. 3) only the shared matrices A, B are reused if more delayed input information is needed. Additionally, if weights are shared more often, then more gradient information are available for learning these weights. As a consequence, potential over-fitting is not so dangerous as in the training of feedforward networks. Or to say it in other words: due to the inclusion of the temporal structure into the network architecture, our approach is applicable to tasks where we have a small training set.

Finally, we discuss a numerical difficulty of the unfolding in time procedure. In typical backpropagation algorithms, the error flow from y_t has to pass the nonlinear transformation tanh many times due to the hidden layers on the way from the past input $u_{t-\tau}$ to y_t. It is well known (see also the chapters of Bengio and Hochreiter) that the error signal decays in such long sequences of transformations $s_{t-\tau}, \cdots, s_t$. The following learning rule eqn (5) overcomes those numerical difficulties which otherwise make it very difficult to find long term inter-temporal structures. For every weight we apply the local adaptation

$$\Delta w_t = -\frac{\eta}{\sqrt{\sum (g_t - g)^2}} g_t \tag{5}$$

with g_t as the gradient from training pattern t and $g = \frac{1}{T}\sum_{t=1}^{T} g_t$ as the averaged gradient over an epoch. In [7] it is shown that this learning rule behaves like a stochastic approximation of a Quasi Newton method. The learning rate is renormalized by the standard deviation of the stochasticity of the error signals. This rescaling avoids the continuous shrinking of the error information through the hidden layer transformations.

3 Overshooting

An obvious generalization of the network in Fig. 3 is the extension of the autonomous recurrence in future direction $t+1, t+2, \cdots$. We call this extension *overshooting* (see Fig. 4). If this leads to good predictions we get as an output a whole sequence of forecasts. Especially for decision support systems, e. g trading system in finance, this additional information is very useful.

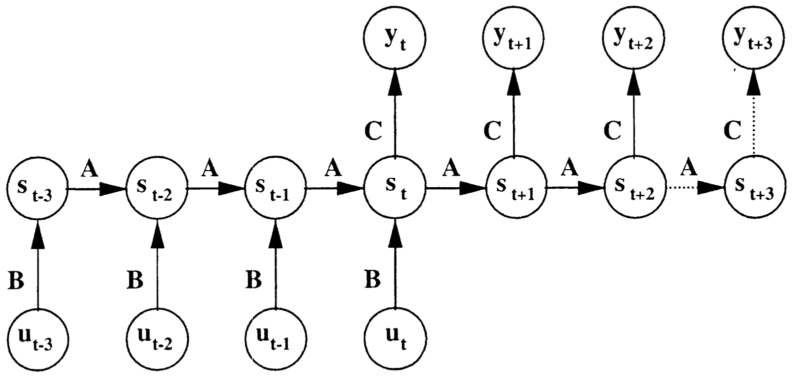

Figure 4: *Overshooting* is the extension of the autonomous part of the dynamics.

In the following we show how overshooting can be realized and we will analyze its properties. First, we discuss how far into the future good predictions can be achieved. We iterate the following: train the model until convergence, if overfitting occurs, include the next output (here y_{t+k+1}) and train it again. Typically we observe the following interesting phenomenon: If the new prediction y_{t+k+1} can be learned by the network, the error for this newly activated time horizon will decrease. But in addition the test error of all the other outputs $y_t, \cdots, y_{t+k}$ will decrease too because more useful information (error signals) is propagated back to the weights. We stop extending the forecast horizon when the error is not longer reducible or even starts to increase.

The most important property of the overshooting network (Fig. 4) is the concatenation of an input driven system and an autonomous system. One may argue that the unfolding in time network (Fig. 3) already consists of recurrent functions and that this recurrent structure has the same modeling effect as the overshooting network. This is definitely not true because the learning algorithm leads to different models for each of these architectures. Backpropagation learning usually tries to model the relationship between the most recent inputs and the output because the fastest adaptation takes place in the shortest path between input and output. Thus, learning mostly focuses on u_t. Only later, learning also extracts useful information from input vectors $u_{t-\tau}$ which are more distant to the output. As a consequence the unfolding in time network (Fig. 3) tries to rely as much as possible on the part of the dynamic which is driven by the most recent inputs $u_t, \cdots, u_{t-\tau}$. In contrast, the overshooting network (Fig. 4) forces the learning by the additional future outputs $(y_{t+1}, \cdots)$ to focus on modeling an internal autonomous dynamic. Overshooting therefore allows us to extend the forecast horizon.

The dimension of s_t in the state transition equation (eqn (1)) represents the *embedding dimension* of an autonomous dynamic subsystem. In such a case, the unfolding network (Fig. 4) is able to extract the correct size of the embedding dimension using pruning techniques. The embedding dimension of a partially autonomous system is typically underestimated by the network architecture of Fig. 3. However, the overshooting network (Fig. 4) learns the correct dimension size because it is forced to learn long term inter-temporal dependencies.

4 Undershooting

This section describes the implementation of a refinement of the time grid by recurrent neural networks. An obvious example is shown in Fig. 5 which can be interpreted as a weekly model using monthly data.

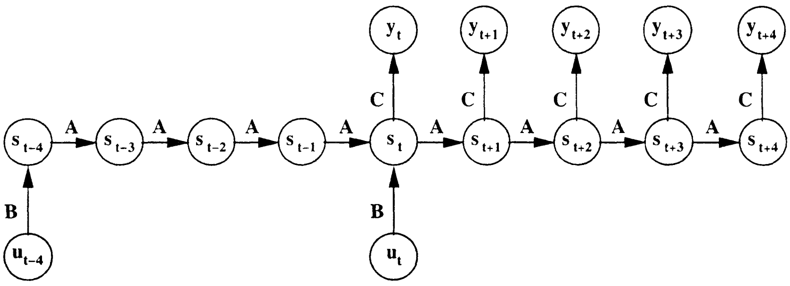

Figure 5: Refinement of the time grid by recurrent neural networks.

While training this network of Fig. 5 only the outputs at time step t and $t+4$ generate error signals because only those targets are available. Due to sharing the output connectors C we are able to compute the other weekly outputs $y_{t+1}, y_{t+2}, y_{t+3}$. We call the usage of a finer time grid for modeling a dynamic system *undershooting.* This technique can generate useful additional information for decision support systems (e. g. automatic trader).

We now propose a special case of undershooting architectures. These models are not focused on the computation of additional output information but are directly related to the considerations on nearly continuous causality.

We illustrate this by an example from economics, but the approach can be applied to any forecast task. Let us assume that we want to forecast a price shift one step ahead $ln(p_{t+1}/p_t)$ (to the first order since $\ln(1+x) \approx x$, this is equivalent to the relative change of p_t). For the moment we ignore the delayed inputs, i. e. all of the input information is now given by u_t. The input u_t is transformed by the matrix B to initialize the recurrent network. If we introduce 5 intermediate time steps in the dynamic systems description we get the architecture of Fig. 6, top. In contrast to our first example in Fig. 5, this network cannot be trained because none of the

Data Mining II, C.A. Brebbia & N.F.F. Ebecken (Editors)
© 2000 WIT Press, www.witpress.com, ISBN 1-85312-821-X

targets $\ln(\frac{p_{t+(k+1)/6}}{p_{t+k/6}}), k = 0, \cdots, 5$ is available.

The redesign leading to the architecture of Fig. 6, bottom, solves this problem of missing targets by exploiting the identity:

$$\ln\left(\frac{p_{t+1}}{p_t}\right) = \sum_{k=0}^{5} \ln\left(\frac{p_{t+(k+1)/6}}{p_{t+k/6}}\right) . \qquad (6)$$

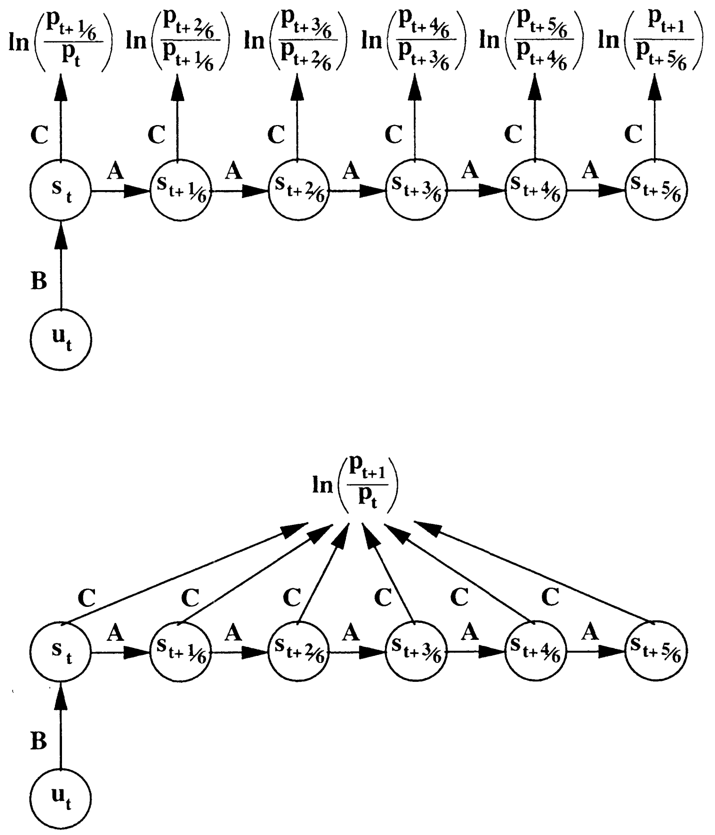
$\ln\left(\frac{p_{t+1/6}}{p_t}\right)$
$\ln\left(\frac{p_{t+2/6}}{p_{t+1/6}}\right)$
$\ln\left(\frac{p_{t+3/6}}{p_{t+2/6}}\right)$
$\ln\left(\frac{p_{t+4/6}}{p_{t+3/6}}\right)$
$\ln\left(\frac{p_{t+5/6}}{p_{t+4/6}}\right)$
$\ln\left(\frac{p_{t+1}}{p_{t+5/6}}\right)$
C
A
B
s_t
$s_{t+1/6}$
$s_{t+2/6}$
$s_{t+3/6}$
$s_{t+4/6}$
$s_{t+5/6}$
u_t
$\ln\left(\frac{p_{t+1}}{p_t}\right)$

Figure 6: The transformation in the lower part allows to train the network even if the intermediate targets are not available.

Assuming finite step sizes, the architecture resembles the way ordinary differential equations are solved. For example, if we want to compute a dynamics at discrete time steps, the finite step forecasting is

$$s_{t+1} = F(s_t) . \qquad (7)$$

Let us further assume, that the dynamic law is given by:

$$\frac{\partial s}{\partial t} = f(s). \tag{8}$$

To bridge the gap between t and $t+1$ we have to integrate the differential equation by:

$$s_{t+1} = s_t + \int_t^{t+1} f(\zeta)d\zeta. \tag{9}$$

This procedure is realized in a similar way by the network architecture in Fig. 6, bottom. First, we describe the dynamics for small time intervals of $1/6$ units. Then, we add up (analog to what an integral does in the limit) all the intermediate shifts of eqn (6) to compute the one step ahead forecast.

We have successfully used this network design with up to 12 intermediate time steps. In such long sequences of hidden layers it is important to use the learning rule eqn (5).

Now, we integrate the delayed inputs which we have ignored so far. If we start again with the network design of Fig. 5, we input into the network rare shocks because intermediate input information is not available. On the other hand if we design the input part analogously to Fig. 6 we loose information by adding up the partial inputs (see Fig. 7). The shared weights B transmit at every intermediate time step only an average information to the recurrent dynamics. However, by our

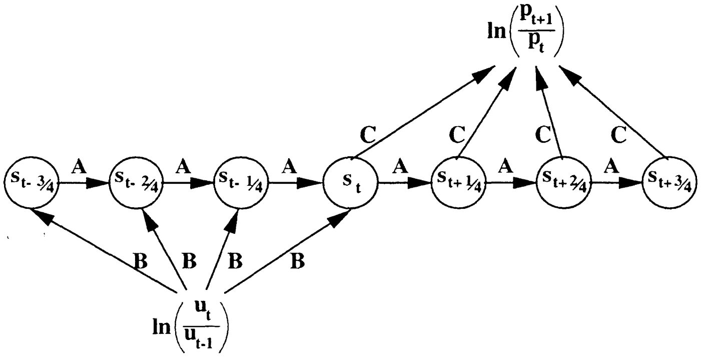

Figure 7: Input design for undershooting networks.

experience the latter approach of Fig. 7 is superior to the approach in Fig. 6. We prefer the use of only one input vector u_t as shown in Fig. 6, bottom, if there are only a few delayed inputs. If one has to use many time lags $\{u_t, u_{t-1}, \ldots\}$, it is more useful to apply the architecture of Fig. 7 instead of integrating the delayed inputs in the state description u_t because this increases the number of weights in B. Indeed, the use of more and more delayed inputs can be achieved with constant number of weights in the network architecture of Fig. 7.

5 Conclusion

We believe that recurrent neural networks supply a very promising framework for the solution of forecasting problems. The key point of building a successful model is the inclusion of structural prior knowledge which can effectively be achieved by extending the network architecture.

Furthermore, this approach allows to integrate many different methods including a nonlinear generalization of *Hidden Markov Models* and to tackle new questions which can not even be expressed without recurrent neural networks.

The described algorithms and architectures are integrated in the *Simulation Environment for Neural Networks*, SENN, a product of Siemens AG. More inforamtion can be found on the web sites `http://www.senn.sni.de` and `http://www.data-mart.de`.

References

[1] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume I: Foundations, pages 318–362. MIT Press/Bradford Books, Cambridge, MA, 1986.

[3] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*. PhD thesis, Harvard University, 1974.

[4] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[5] S. Haykin. *Neural Networks. A Comprehensive Foundation*. Macmillan College Publishing, New York, 1994. second edition 1998.

[6] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[7] Ralph Neuneier and Hans Georg Zimmermann. How to Train Neural Networks. In *Neural Networks: Tricks of the Trade*, pages 373–423. Springer Verlag, Berlin, 1998.